# Complexity and Accumulators

Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University

# Today's goals

- Accounting for *cost* of computation (complexity)
- Accumulating "history" using *accumulators*

# Example: Partial Sums

```
;; sums: (listOf number) -> (listOf number)
;; (sums alon) computes the partial sums for n; it returns a list of
;; numbers, psum, such that the ith element of psum is the sum of the
;; numbers preceding (and including) the ith element of alon e.g.,
;; (sums '(1 2 3 4 5)) = '(1 3 6 10 15)

(define (sums alon)
  (cond [(empty? alon) empty]
        [else
         (cons (first alon)
            (map (lambda (x) (+ x (first alon)))
                 (sums (rest alon))))]))
```

# Question: how many additions does function sums perform?

Reduction sequence:

…(list 5)… => . . . =>

…(list 4 (+ 5 4))… =>

…(list 4 9)… => . . . =>

…(list 3 (+ 4 3) (+ 9 3))… => . . . =>

…(list 3 7 12)… => . . . =>

…(list 2 (+3 2) (+7 2) (+12 2))… => . . . =>

…(list 2 5 9 14)… => . . . =>

…(list 1 (+2 1) (+5 1) (+9 1) (+14 1))… => . . . =>

(list 1 3 6 10 15)

# Cost accounting

- Measure computation cost in reduction steps using our reduction semantics. Models actual cost reasonably well.
- Consider three algorithms
  - Cost-A(n) = $2*n^3 + n^2 + 50$
  - Cost-B(n) = $3*n^2 + 100$
  - Cost-C(n) = $2^n$
- Which algorithm is best?
- Which algorithm works best for large n?
- Can we formalize this notion?

# Order of Complexity

- We'll say that Cost-X is "order $f$(n))", or simply "$O(f$(n))" (read "Big-O of $f$(n))" ) if
  - Cost-X(n) < factor * $f$(n) for sufficiently large n
- Examples:
  - Cost-A(n) = $2*n^3 + n^2 + 1$  Cost-A is $O(n^3)$
  - Cost-B(n) = $3*n^2 + 10$  Cost-B is $O(n^2)$
  - Cost-C(n) = $2^n$  Cost-C is $O(2^n)$

# Famous "Complexity Classes"

- $O(1)$ constant-time (head, tail)
- $O(\log n)$ logarithmic (binary search)
- $O(n)$ linear (vector multiplication)
- $O(n * \log n)$ "n log n" (sorting)
- $O(n^2)$ quadratic (matrix addition)
- $O(n^3)$ cubic (matrix multiplication)
- $n^{O(1)}$ polynomial (…many! …)
- $2^{O(n)}$ exponential (guess password)

# Improving Performance

- The sums function performs n*(n-1)/2 additions to compute partial sums for a list of n numbers
- We can do much better than $O(n^2)$!
- What information do we need to do better?
  - This is basically the “lost history” in the recursive call

# Accumulator version of same program

- Idea: as the list is successively decomposed into **first** and **rest**, the **sums** function can accumulate the sum of the numbers to the left of **rest**.
- Template Instantiation:

```
(define (sums-help lon sum)
  (cond [(empty? lon) … ]
        [else … (first lon) … sum …
              (sums-help (rest lon) ..) ]))
```

# Accumulator version of same program

```
;; sums-help: (listOf number) number -> (listOf number)
;; Invariant: sum is the sum of the numbers that preceded alon in alon0
(define (sums-help alon sum)
  (cond
    [(empty? alon) empty]
    [else
      (local [(define new-sum (+ sum (first l)))]
        (cons new-sum (sums-help (rest l) new-sum)))]))

;; sums: (listOf number) -> (listOf number)
(define (sums alon0) (sums-help alon0  0))
```

# Question: how many additions does the accumulator version perform?

Reduction sequence:

(sums-help (list 1 2 3 4 5) 0) => . . . =>

…(+ 0 1)… => . . . =>

(cons 1 (sums-help (list 2 3 4 5) 1)) => . . . =>

…(+ 1 2)… => . . . =>

(cons 1 (cons 3 (sums-help (list 3 4 5) 3))) => . . . =>

…(+ 3 3)… => . . . =>

(cons 1 (cons 3 (cons 6 (sums-help (list 4 5) 6)))) => . . . =>

…(+ 6 4)… => . . . =>

(cons 1 (cons 3 (cons 6 (cons 10 (sums-help (list 5) 10))))) => . . . =>

…(+ 10 5)… => . . . =>

(cons 1 (cons 3 (cons 6 (cons 10 (cons 15 empty)))))

# Formulating an Accumulator

- If we decide to use an accumulator, we need to answer three questions:
  - What should the initial value for the accumulator be?
  - How will we modify the accumulator in each recursive call? (What will we "accumulate"?)
  - How will we use the accumulator to produce the final result?

# Naïve List Reversal

```
(define (rev l)
  (cond [(empty? l) empty]
        [else (append (rev (rest l))
                      (list (first l)))]))
```

# Reversal using an accumulator

```
;; Invariant: ans is the reversed list of all items
;; that preceded l in l0

(define (rev-help l ans)
  (cond [(empty? l) ans]
        [else (rev-help (rest l) (cons (first l) ans))]))

(define (fast-rev l0)  (rev-help l empty))
```

# Added Expressivity

- Code simplification using accumulators
- Consider the list reverse function
  - Takes '(1 2 3 4 5) and produces '(5 4 3 2 1)
- How did we write this function in the naïve version? Used append. Ugh.
- What information did we use to do better?
  - This is basically the "lost history" of the recursive call
- Is this list reversal example really different from the list accumulation example?

# Naïve List Flattening

- ;; (flatten: (genListOf symbol) -> (listOf symbol))
  ;; (flatten agl) returns a list of the symbols in order of appearance
  ;; (flatten '((a b) c ((d))) = '(a b c d)
- ```
  (define (flatten agl)
    (cond [(empty? agl) empty]
          [else (local [(define head (first agl))
                        (define tail (flatten (rest agl)))]
                  (cond [(empty? head)  tail]
                        [(cons? head)  (append (flatten head) tail)]
                        [else  (cons head tail)]))]))
  ```
- Note: we wrote this function so that the symbol type can be replaced by any non-list type.
  ```

# Accumulator version

;; flatten-help: (genListOf symbol) (listOf symbol) -> (listOf symbol)
;; (flatten agl los) returns a list of the symbols in agl appended to los
;; (flatten '((a b) c ((d)) '(e)) = '(a b c d e)

;; What is the invariant for the accumulator variable los?

```
(define (flatten-help agl los)
  (cond [(empty? agl) los]
        [else (local [(define head (first agl))
                      (define tail (flatten-help (rest agl) los))]
                (cond [(empty? head)  tail]
                      [(cons? head)  (flatten-help head tail)]
                      [else  (cons head tail)]))]))
```